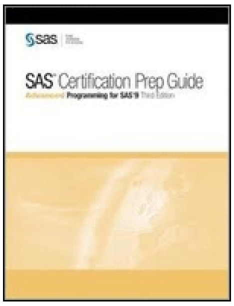


SAS Certification Prep Guide: Advanced Programming for SAS 9, Third Edition

by SAS Institute

Chapter 22: Using Best Practices

Overview

Introduction

This chapter demonstrates different ways of using best SAS programming practices to optimize performance. As you compare the techniques described in this chapter, remember that differences in the use of resources are affected by which operating environment you work in and by the characteristics of your data.

This chapter is organized by topics that emphasize the following basic principles:

- Execute only necessary statements.
- Eliminate unnecessary passes of the data.
- Read and write only the data that you require.
- Store data in SAS data sets.
- Avoid unnecessary procedure invocation.

Each topic includes comparative examples that can improve the efficiency of your programs. Write programs to generate your own benchmarks, and adopt the programming techniques that produce the most savings for you.

> **Note** This chapter does not cover the SAS Scalable Performance Data Engine (SAS SPD Engine), which is a SAS 9.1 technology for threaded processing. For details about using the SAS SPD Engine to improve performance, see the SAS documentation.

Objectives

In this chapter, you learn to efficiently

- subset observations
- create new variables
- process and output data conditionally
- create multiple output data sets and sorted subsets
- modify variable attributes
- select observations from SAS data sets and external files
- subset variables
- read data from SAS data sets
- invoke SAS procedures.

Prerequisites

Before beginning this chapter, you should complete the following chapters:

- Part 1: SQL Processing with SAS
  - "Performing Queries Using PROC SQL" on page 4
  - "Performing Advanced Queries Using PROC SQL" on page 29
  - "Combining Tables Horizontally Using PROC SQL" on page 86

Executing Only Necessary Statements

Overview

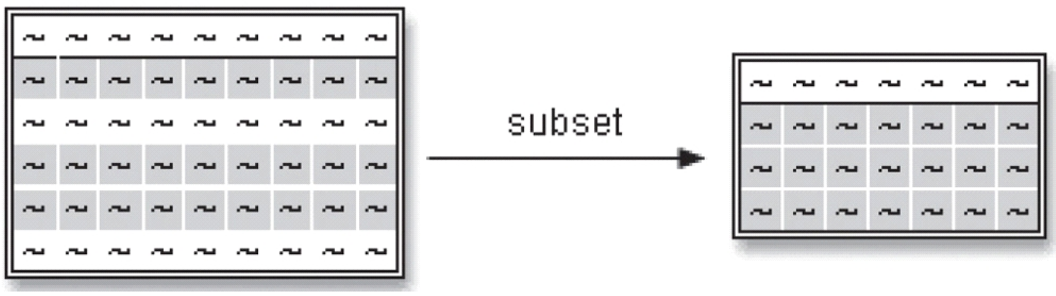
Best practices specify that you should write programs that cause SAS to execute only necessary statements. When you execute the minimum number of statements in the most efficient order, you minimize the hardware resources that SAS uses. The resources that are affected include disk usage, memory usage, and CPU usage.

Techniques for executing only the statements you need include

- subsetting your data as soon as is logically possible
- processing your data conditionally by using the most appropriate syntax for your data.

Positioning of the Subsetting IF Statement

To subset your data based on a newly derived or computed variable, you must use the subsetting IF statement in a DATA step. You can subset output data to a SAS data set by processing only those observations that meet a specified condition.



The subsetting IF statement causes the DATA step to continue processing only those raw data records or observations that meet the condition of the expression specified in the IF statement. The resulting SAS data set or data sets contain a

subset of the original external file or SAS data set.

Position the subsetting IF statement in the program so that it checks the subsetting condition as soon as it is logically possible and so that unnecessary statements do not execute. When a statement is false, no further statements are processed for that observation.

Also, remember to subset data before performing calculations and to minimize the use of function calls or arithmetic operators. Unnecessary processing of unwanted observations results in higher expenditure of hardware resources.

Comparative Example: Creating a Subset of Data

Overview

Suppose you want to create a subset of data, calculate six new variables, and conditionally output data by reading from the SAS data set *Retail.Order_fact*. The data set should contain new variables for

- the month of the order
- the elapsed time between the order date and the delivery date
- the profit, based on the retail price, discount, and unit price
- total profit
- total discount
- total wait time.

The subset of data that includes only orders for the month of December is approximately 9.66% of the data.

You can accomplish this task by using a subsetting IF statement. Placement of this statement in the DATA step can affect the efficiency of the DATA step in terms of CPU time and real time. Notice the comparison between these two approaches:

1. A Subsetting IF Statement at the Bottom
2. A Subsetting IF Statement near the Top.

The following sample programs show each of these techniques. You can use these samples as models for creating benchmark programs in your own environment. Your results might vary depending on the structure of your data, your operating environment, and the resources that are available at your site. You can also view general recommendations for executing only necessary statements.

Programming Techniques

---

❶ A Subsetting IF Statement at the Bottom

This program calculates six new variables before the subsetting IF statement selects only observations whose values for Month are *12*.

```
data profit;
   retain TotalProfit TotalDiscount TotalWait Count 0;
   set retail.order_fact;
   MonthOfOrder=month(order_date);
   WaitTime=sum(delivery_date,-order_date);
   if discount gt . then
      CalcProfit=sum((total_retail_price*discount),-costprice_per_unit)
                 *quantity;
      else CalcProfit=sum(total_retail_price,-costprice_per_unit)
                 *quantity;
      TotalProfit=sum(totalprofit,calcprofit);
      TotalDiscount=sum(totaldiscount,discount);
      TotalWait=sum(totalwait,waittime);
      Count+1;
      if monthoforder=12;
run;
```

❷ A Subsetting IF Statement near the Top

In this program, the subsetting IF statement is positioned immediately after the value for **MonthofOrder** has been calculated. If the value is not *12*, then no further statements are processed for that observation. In this program, calculations are performed on a smaller number of observations, which results in greater program efficiency.

```
data profit;
   retain TotalProfit TotalDiscount TotalWait Count 0;
   set retail.order_fact;
   MonthOfOrder=month(order_date);
   if monthoforder=12;
   WaitTime=sum(delivery_date,-order_date);
   if discount gt . then
      CalcProfit=sum((total_retail_price*discount),-costprice_per_unit)
                 *quantity;
      else CalcProfit=sum(total_retail_price,-costprice_per_unit)
                 *quantity;
      Total Profit=sum(totalprofit, calcprofit);
      TotalDiscount=sum(totaldiscount,discount);
      TotalWait=sum(totalwait,waittime);
      Count+1;
run;
```

General Recommendations

- Position the subsetting IF statement in a DATA step as soon as is logically possible in order to save the most resources.

In the last comparative example you saw how a subsetting IF statement can be positioned in the DATA step so that no further statements are processed for that observation. Next, look at how different programming techniques can be used to

- create variables conditionally using DO groups
- create variables conditionally when calling functions.

Before viewing the sample code for these two comparative examples, review guidelines for using these techniques.

Using Conditional Logic Efficiently

You can use conditional logic to change how SAS processes selected observations. Two techniques—IF-THEN/ELSE statements and SELECT statements—can be used interchangeably and perform comparably. Based on the characteristics of your data and depending on your environment, one of these techniques might give you better performance. Choose a technique that conserves your programming time and makes the program easiest to read.

Technique	Action
IF-THEN/ELSE statement	executes a SAS statement for observations that meet specific conditions.
SELECT statement	executes one of several statements or groups of statements.

Note The number of conditions (values) tested and the type of variable tested affect CPU resources.

Use IF-THEN/ELSE statements when

- the data values are character values
- the data values are not uniformly distributed
- there are few conditions to check.

For best practices, follow these guidelines for writing efficient IF/THEN logic:

- For mutually exclusive conditions, use the ELSE IF statement rather than an IF statement for all conditions except the first.

- Check the most frequently occurring condition first, and continue checking conditions in descending order of frequency.
- When you execute multiple statements based on a condition, put the statements in a DO group.

Use SELECT statements when

- you have a long series of mutually exclusive conditions
- data values are uniformly distributed.

Before writing conditional logic, determine the distribution of your data values. You can use the

- FREQ procedure to examine the distribution of the data values
- GCHART or GPLOT procedure to display the distribution graphically
- UNIVARIATE procedure to examine distribution statistics and to display the information graphically.

Comparative Example: Creating Variables Conditionally Using DO Groups

Overview

Suppose you want to calculate an adjusted profit based on the values of the variable **`Order_Type`** in the data set *Retail.Order_fact*. For retail sales, which are represented by the value 1, the adjusted profit should be calculated as 105% of profit. For catalog sales, which are represented by the value *2*, the adjusted profit should be calculated as 103% of profit. For Internet sales, which are represented by the value *3*, the adjusted profit should be equal to profit.

The following table shows that the values for the variable **`Order_Type`** are not uniformly distributed.

Order Type		
Order_Type	**Frequency**	**Percent**
1	3579850	75.23
2	635645	13.36
3	542850	11.41

The following table shows that the values for the variable **`Discount`** also are not uniformly distributed.

Discount		
Discount	Frequency	Percent
.	4712585	99.04
30%	19740	0.41
40%	15170	0.32
50%	9805	0.21
60%	1045	0.02

Techniques for creating new variables conditionally include

1. IF-THEN/ELSE statements
2. SELECT statements.

The following sample programs show each of these techniques. You can use these samples as models for creating benchmark programs in your own environment. Your results might vary depending on the structure of your data, your operating environment, and the resources that are available at your site. You can also view general recommendations for executing only necessary statements.

Programming Techniques

---

❶ IF-THEN/ELSE Statements

This program uses IF-THEN/ELSE statements with DO groups to conditionally execute multiple statements that calculate an adjusted profit. Conditions are checked in descending order of frequency.

```
data retail.order_info_1;
   set retail.order_fact;
   if order_type=l then
      do;                    /* Retail Sale */
         Float=delivery_date-order_date;
         RevenueQuarter=qtr(order_date);
         AveragePrice=total_retail_price/quantity;
         if discounts=. then NetPrice=total_retail_price;
         else NetPrice=total_retail_price-discount;
         Profit=netPrice-(quantity*costprice_per_unit)*1.05;
      end;
   else if order_type=2 then
      do;                   /* Catalog Sale */
         Float=delivery_date-order_date;
         RevenueQuarter=qtr(order_date);
         AveragePrice=total_retail_price/quantity;
         if discounts=. then NetPrice=total_retail_price;
         else NetPrice=total_retail_price-discount;
         Profit=netprice- (quantity*costprice_per_unit)*1.03;
      end;
```

```
   else
      do;                    /* Internet Sale */
         Float=delivery_date-order_Date;
         RevenueQuarter=qtr(order_date);
         AveragePrice=total_retail_price/quantity;
         if discount=. then HetPrice=total_retail_price;
         else NetPrice=total_retail_price-discount;
         Profit=netprice- (quantity*costprice_per_unit);
      end;
run;
```

❷ SELECT Statements

This program uses SELECT/WHEN statements with DO groups to conditionally execute multiple statements that calculate an adjusted profit. Conditions are checked in descending order of frequency.

```
data retail.order_info_2;
   set retail.order_fact;
   select(order_type);
     when (1)
        do;                                /* Retail Sale */
          Float=delivery_date-order_date;
          RevenueQuarter=qtr(order_date);
          AveragePrice=total_retail_price/quantity;
          if discount=. then NetPrice=total_retail_price;
          else NetPrice=total_retail_price-discount;
          Profit=netprice-(quantity*costprice_per_unit)*1.05;
        end;
     when (2)
        do;                                /* Catalog Sale */
           Float=delivery_date-order_date;
           RevenueQuarter=qtr(order_date);
           AveragePrice=total_retail_price/quantity;
           if discount=. then NetPrice=total_retail_price;
           else NetPrice=total_retail_price-discount;
           Profit=netprice-(quantity*costprice_per_unit)*1.03;
        end;
     otherwise
        do;                                /* Internet Sale */
           Float=delivery_date-order_date;
           RevenueQuarter=qtr(order_date);
           AveragePrice=total_retail_price/quantity;
           if discount=. then NetPrice=total_retail_price;
           else NetPrice=total_retail_price-discount;
           Profit=netprice-(quantity*costprice_per_unit);
        end;
   end;
run;
```

General Recommendations

- Check the most frequently occurring condition first, and continue checking conditions in descending order of frequency, regardless of whether you use IF-THEN/ELSE or SELECT statements.
- When you execute multiple statements based on a condition, put the statements in a DO group.

Comparative Example: Creating Variables Conditionally When Calling Functions

Overview

Suppose you want to create a report that includes a new variable that is based on the value of an existing variable in the SAS data set *Retail.Order_fact*. Values for the new **`Month`** variable are extracted from the existing variable **`Order_Date`** by using the MONTH function.

The following table shows that the values for **`Month`** are fairly evenly distributed.

Month of Order		
Month	**Frequency**	**Percent**
Jan	386535	8.12
Feb	319895	6.72
Mar	350255	7.36
Apr	421265	8.85
May	443535	9.32
Jun	429760	9.03
Jul	438085	9.21
Aug	443075	9.31
Sep	299260	6.29
Oct	373400	7.85
Nov	393750	8.27
Dec	459530	9.66

Techniques for creating new variables conditionally include

1. Parallel IF Statements
2. ELSE IF Statements, Many Function References
3. ELSE IF Statements, One Function Reference
4. SELECT Group.

The following sample programs show each of these techniques. You can use these samples as models for creating benchmark programs in your own environment. Your results might vary depending on the structure of your data, your operating environment, and the resources that are available at your site. You can also view general recommendations for executing only necessary statements.

Programming Techniques

---

❶ Parallel IF Statements

This program calls the MONTH function 12 times. With these non-exclusive cases, each IF statement executes for each observation that is read from *Retail.Order_Fact*. This is the least efficient approach.

```
data retail.orders;
   set retail.order_fact;
   if month(order_date)=1 then Month='Jan';
   if month(order_date)=2 then Month='Feb';
   if month(order_date)=3 then Month='Mar';
   if month(order_date)=6 then Month='Jun';
   if month(order_date)=7 then Month='Jul';
   if month(order_date)=8 then Month='Aug';
   if month(order_date)=9 then Month='Sep';
   if month(order_date)=10 then Month='Oct';
   if month(order_date)=11 then Month='Nov';
   if month(order_date)=12 then Month='Dec';
run;
```

❷ ELSE IF Statements, Many Function References

This program uses ELSE IF statements that call the function MONTH. Once the true condition is found, subsequent ELSE IF statements are not executed. This is more efficient than using parallel IF statements, but the MONTH function is executed many times.

```
data retail.orders;
   set retail.order_fact;
   if month(order_date)=1 then Month='Jan';
   else if month(order_date)=2 then Month='Feb';
   else if month(order_date)=3 then Month='Mar';
   else if month(order_date)=4 then Month='Apr';
   else if month(order_date)=5 then Month='May';
   else if month(order_date)=6 then Month='Jun';
   else if month(order_date)=7 then Month='Jul';
   else if month(order_date)=10 then Month='Oct';
   else if month(order_date)=11 then Month='Nov';
   else if month(order_date)=12 then Month='Dec';
run;
```

❸ ELSE IF Statements, One Function Reference

This program uses the MONTH function to find the value of **`Order_Date`**, but only once. The MONTH function is called immediately after reading the data set and before any IF-THEN/ELSE statements execute. This is efficient.

```
data retail.orders(drop=mon);
   set retail.order_fact;
   mon=month(order_date);
   if mon=l then Month='Jan';
   else if mon=2 then Month='Feb';
   else if mon=3 then Month='Mar';
   else if mon=4 then Month='Apr';
   else if mon=5 then Month='May';
   else if mon=6 then Month='Jun';
   else if mon=7 then Month='Jul';
   else if mon=8 then Month='Aug';
   else if mon=9 then Month='Sep';
   else if mon=10 then Month='0ct';
   else if mon=ll then Month='Nov';
   else if mon=12 then Month='Dec';
run;
```

❹ SELECT Group

In this program, the SELECT statement calls the MONTH function only once, before WHEN statements execute and assign values for **`Month.`** This is efficient.

```
data retail.orders;
   set retail.order_fact;
   select(month(order_date));
     when (1) Month= Jan';
     when (2) Month= Feb';
     when (3) Month= Mar';
     when (4) Month= Apr';
     when (5) Month= May';
     when (6) Month= Jun';
     when (7) Month= Jul';
     when (8) Month= Aug';
     when (11) Month 'Nov';
     when (12) Month 'Dec';
     otherwise;
  end;
run;
```

General Recommendations

- Avoid using parallel IF statements, which use the most resources and are the least efficient way to conditionally execute statements.
- Use IF-THEN/ELSE statements and SELECT blocks to be more efficient.

- To significantly reduce the amount of resources used, write programs that call a function only once instead of repetitively using the same function in many statements. SAS functions are convenient, but they can be expensive in terms of CPU resources.

You have seen different ways of efficiently creating variables conditionally, which reinforce the principle of executing only necessary statements. The next comparative example in this topic addresses techniques that can be used to create data in DO groups.

Before viewing this example, you might want to review the following information about which statements are needed in DO groups.

Using DO Groups Efficiently

You can conditionally execute only necessary statements by placing them in DO groups that are associated with IF-THEN/ELSE statements or with SELECT/WHEN statements. Groups of statements execute only when a particular condition is true. Remember to use the following criteria when choosing which technique is more efficient:

	IF-THEN/ELSE Statements	SELECT/WHEN Statements
The number of conditions	few	many
The distribution of a variable's values	not uniform	uniform

When using a DO group with IF-THEN/ELSE statements, add DO after the THEN clause, and add an END statement after all of the statements that you want executed as a group.

```
data orders;
   set company.orders;
   if order_type = 1 then
      do;
      <multiple executable statements here>
      end;
   else if order_type = 2 then
      do;
      <multiple executable statements here>
      end;
   else if order_type = 3 then
      do;
      <multiple executable statements here>
      end;
run;
```

Note Use an IF-THEN DO group when you create multiple variables based on a condition.

When using a DO group with SELECT/WHEN statements, add DO after the WHEN condition, and add an END statement after all of the statements that you want executed as a group. Use an OTHERWISE statement to specify the statements that you want executed if no WHEN condition is met.

```
data orders;
   set company.orders;
   select (order_type);
      when (1)
        do;
        <multiple executable statements here>
        end;
      when (2)
        do;
        <multiple executable statements here>
        end;
      when (3)
        do;
        <multiple executable statements here>
        end;
      otherwise;
   end;
run;
```

Remember that IF-THEN/ELSE and SELECT/WHEN logic require that there be no intervening statements between the IF and the ELSE conditions or between the SELECT and the WHEN conditions.

Comparative Example: Creating Data in DO Groups

Overview

Suppose you want to identify which customer groups are Club Members, Club Gold Members, or Internet/Catalog members, based on data from the data set *Retail.Customer_hybrid*. You also want to identify the nature of customer activity as "inactive", "low activity", "medium activity", or "high activity".

The following table shows the distribution of values for **`Customer_Type_ID`**.

Customer Type ID		
Customer_Type_ID	Frequency	Percent
1010	568446	12.39
1020	579156	12.62
1030	571608	12.46
1040	574209	12.52
2010	566457	12.35
2020	571914	12.47
2030	579054	12.62
3010	576810	12.57

Techniques for creating new variables based on the values of specific variables include

1. Serial IF Statements
2. SELECT, IF/SELECT Statements
3. Nested SELECT Statements
4. IF-THEN/ELSE IF Statements with a Link.

The following sample programs show each of these techniques. You can use these samples as models for creating benchmark programs in your own environment. Your results might vary depending on the structure of your data, your operating environment, and the resources that are available at your site. You can also view general recommendations for executing only necessary statements.

Programming Techniques

---

❶ Serial IF Statements

This program creates a permanent SAS data set named *Retail.Customers* by reading the *Retail.Customer_hybrid* data set. Serial IF statements are used to populate the variables **`Customer_Group`** and **`Customer_Activity`**.

```
data retail.customers;
```

```
   length Customer_Group $ 26 Customer_Activity $ 15;
   set retail.customer_hybrid;
   if substr(put(customer_type_ID,4.),1,2)='10' then
      customer_group='Orion Club members';
   if substr(put(customer_type_ID,4.),1,2)='20' then
      customer_group='Orion Club Gold members';
   if substr(put(customer_type_ID,4.),1,2)='30' then
      customer_group='Internet/Catalog Customers';
   if substr(put(customer_type_ID,4.),1,2) in ('10', '20') and
      substr(put(customer_type_ID,4.),3,2)='10' then
         customer_activity='inactive';
   if substr(put(customer_type_ID,4.),1,2) in ('10', '20') and
      substr(put(customer_type_ID,4.),3,2)='20' then
         customer_activity='low activity';
   if substr(put(customer_type_ID,4.),1,2) in ('10', '20') and
      substr(put(customer_type_ID,4.), 3, 2) = '30' then
         customer_activity='medium activity';
   if substr(put(customer_type_ID,4.),1,2) in ('10', '20') and
      substr (put (customer_type_ID, 4.),3,2) = '40' then
         customer_activity='high activity';
run;
```

❷ SELECT, IF/SELECT Statements

This program creates a permanent SAS data set named *Retail.Customers* by reading the *Retail.Customer_hybrid* data set. SELECT/WHEN logic and SELECT/WHEN statements in an IF/THEN DO group populate the variables **Customer_Group** and **Customer_Activity**. If the value of the first two digits of **Customer_Type_ID** is *10, 20*, or *30*, then **Customer_Group** is populated. If the value of the first two digits of **Customer_Type_ID** is *10* or *20*, then **Customer_Activity** is populated by reading the last two digits of **Customer_Type_ID.**

```
data retail.customers;
   length Customer_Group $ 26 Customer_Activity $ 15;
   set retail.customer_hybrid;
     select(substr(put(customer_type_ID,4.),1,2));
        when ('10') customer_group='Orion Club members';
        when ('20') customer_group='Orion Club Gold members';
        when ('30') customer_group='Internet/Catalog Customers';
        otherwise;
     end;
   if substr(put(customer_type_ID,4.),1,2) in ('10', '20') then
     do;
        select(substr(put(customer_type_ID,4.),3,2));
           when ('10') customer_activity='inactive';
           when ('20') customer activity= 'low activity';
           when ('30') customer activity= 'medium activity';
           when ('40') customer activity= 'high activity';
           Otherwise;
        end;
     end;
run;
```

❸ Nested SELECT Statements

This program creates a permanent SAS data set named *Retail.Customers* by reading the *Retail.Customer_hybrid* data set. Nested SELECT statements are used to populate the variables **Customer_Group** and **Customer_Activity.**

```
data retail.customers;
   length Customer_Group $ 26 Customer_Activity $ 15;
   set retail.customer_hybrid;
   select(substr(put(customer_type_ID,4.),1,2));
     when ('10')
       do;
          customer_group='Orion Club members';
          select(substr(put(customer_type_ID,4.),3,2));
             when ('10')  customer activity= 'inactive';
             when ('20')  customer activity= 'low activity';
             when ('30')  customer activity= 'medium activity'
             when ('40')  customer activity= 'high activity';
```

```
            otherwise;
      end;
    end;
    when ('20')
       do;
          customer_group='Orion Club Gold members';
          select(substr(put(customer_type_ID,4.),3,2));
             when ('10')  customer activity= 'inactive';
             when ('20')  customer activity= 'low activity';
             when ('30')  customer activity= 'medium activity';
             when ('40')  customer activity= 'high activity';
             otherwise;
           end;
        end;
      when ('30') customer_group='Internet/Catalog Customers';
      otherwise;
    end;
run;
```

4 IF-THEN/ELSE IF Statements with a Link

This program creates a permanent SAS data set named *Retail.Customers* by reading the *Retail.Customer_hybrid* data set. IF-THEN/ELSE IF statements are used with a link to populate the variables **Customer_Group** and **Customer_Activity.**

```
data retail.customers;
   length Customer_Group $ 26 Customer_Activity $ 15;
   set retail.customer_hybrid;
   if substr(put(customer_type_ID,4.),1,2)='10' then
      do;
         customer_group='Orion Club members';
         link activity;
      end;
   else if substr(put(customer_type_ID,4.), 1,2) = '20' then
      do;
         customer_group='Orion Club Gold members';
         link activity;
      end;
   else if substr(put(customer_type_ID,4.),1,2)='30' then
         customer_group='Internet/Catalog Customers';
   return;
   activity:
   if substr(put(customer_type_ID,4.),3,2)='10' then
         customer_activity='inactive';
   else if substr(put(customer_type_ID,4.I,3,2)='20' then
         customer_activity='low activity';
   else if substr(put(customer_type_ID,4.),3,2)='30' then
         customer_activity='medium activity';
   else if substr(put(customer_type_ID,4.),3,2)='40' then
         customer_activity='high activity';
   return;
run;
```

General Recommendations

- Avoid serial IF statements because they use extra resources.

Eliminating Unnecessary Passes through the Data

Best practices specify that you should eliminate unnecessary passes through the data. To minimize I/O operations and CPU time, avoid reading or writing data more than necessary. Accomplish this by taking advantage of one-step processing, which can lead to efficiencies.

Using a Single DATA or PROC Step to Enhance Efficiency

Whenever possible, use a single DATA or PROC step to enhance efficiency. Techniques that minimize passes through the

data include

- using a single DATA step to create multiple output data sets
- using the SORT procedure with a WHERE statement to create sorted subsets
- using the DATASETS procedure to modify variable attributes.

Before viewing comparative examples that address these techniques, it might be helpful to review the following information about these practices.

Using a Single DATA Step to Create Multiple Output Data Sets

It is good programming practice to take advantage of the DATA step's ability to create multiple output data sets at the same time. This can be more efficient than using a series of individual DATA steps. Using a single DATA step saves resources because input data is read only once.

Comparative Example: Creating Multiple Subsets of a SAS Data Set

Overview

Suppose you want to create five subsets of data from the data set *Retail.Customer*. You need a subset for each of five countries. Techniques for creating multiple subsets include writing

1. Multiple DATA Steps
2. A Single DATA Step.

The following sample programs show each of these techniques. You can use these samples as models for creating benchmark programs in your own environment. Your results might vary depending on the structure of your data, your operating environment, and the resources that are available at your site. You can also view general recommendations for eliminating unnecessary passes through the data.

Programming Techniques

---

❶ Multiple DATA Steps

This program includes multiple DATA steps and subsequently reads data five times from the same *Retail.Customer* data set. Individual subsetting IF statements appear in five separate DATA steps.

```
data retail.UnitedStates;
   set retail.customer;
   if country='US';
run;

data retail.France;
   set retail.customer;
   if country='FR';
run;

data retail.Italy;
   set retail.customer;
   if country='IT';
run;

data retail.Germany;
   set retail.customer;
   if country='DE';
run;

data retail.Spain;
   set retai 1.customer;
   if country='ES';
run;
```

❷ A Single DATA Step

This program uses only one DATA step to create five output data sets. The data set *Retail.Customer* is read only once. Also, IF-THEN/ELSE statements are used to conditionally output data to specific data sets.

```
data retail.UnitedStates
     retail.France
     retail.Italy
     retail.Germany
     retail.Spain;
   set retail.customer;
   if country='US' then output retail.UnitedStates;
   else if country='FR' then output retail.France;
   else if country='IT' then output retail.Italy;
   else if country='DE' then output retail.Germany;
   else if country='ES' then output retail.Spain;
run;
```

General Recommendations

- When creating multiple subsets from a SAS data set, use a single DATA step with IF-THEN/ELSE IF logic to output to appropriate data sets.
- Because data can be cached in Windows and UNIX, there might not be dramatic I/O differences between the two programs.

Using the SORT Procedure with a WHERE Statement to Create Sorted Subsets

It is good programming practice to take advantage of the SORT procedure's ability to sort and subset in the same PROC step. This is more efficient than using two separate steps to accomplish this—a DATA step to subset followed by a procedure step that sorts.

Comparative Example: Creating a Sorted Subset of a SAS Data Set

Overview

Suppose you want to create a sorted subset of a SAS data set named *Retail.Customer*. You want only data for customers in the United States, France, Italy, Germany, and Spain.

Techniques for creating sorted subsets of SAS data sets include

1. A DATA Step and PROC SORT
2. PROC SORT with a WHERE Statement.

The following sample programs show each of these techniques. You can use these samples as models for creating benchmark programs in your own environment. Your results might vary depending on the structure of your data, your operating environment, and the resources that are available at your site. You can also view general recommendations for eliminating unnecessary passes through the data.

Programming Techniques

❶ A DATA Step and PROC SORT

This program has two steps. The first step creates a SAS data set by subsetting observations based on the value of the variable **Country.** The second step sorts the data according to the values for each country. Passing through all the data once and the subset twice can increase I/O and CPU operations.

```
data retail.CountrySubset;
   set retail.customer;
   where country in('US','FR','IT','DE','ES');
run;

proc sort data=retail.CountrySubset;
   by country;
run;
```

❷ PROC SORT with a WHERE Statement

In one step, this program sorts data and selects only those observations that meet the conditions of the WHERE statement. Processing only one data set once saves CPU and I/O resources.

Note that if this program did not create a second data set named *Retail.CountrySubset*, it would write over the data set named *Retail.Customer* with only part of the data.

```
proc sort data=retail.customer out=retail.CountrySubset;
   by country;
   where country in('US','FR','IT','DE','ES');
run;
```

General Recommendations

- When you need to process a subset of data with a procedure, use a WHERE statement in the procedure instead of creating a subset of data and reading that data with the procedure.
- Write one program step that both sorts and subsets. This approach can take less programmer time and debugging time than writing separate program steps that subset and sort.

Using the DATASETS Procedure to Modify Variable Attributes

Use PROC DATASETS instead of a DATA step to modify data attributes. The DATASETS procedure uses fewer resources than the DATA step because it processes only the descriptor portion of the data set, not the data portion. PROC DATASETS retains the sort flag, as well as indexes.

> **Note** You cannot use the DATASETS procedure to modify the type, length, or position of variables because these attributes directly affect the data portion of the data set. To perform these operations, use the DATA step.

Comparative Example: Changing the Variable Attributes of a SAS Data Set

Overview

Suppose you want to change the variable attributes in *Retail.NewCustomer* to make them consistent with those in the *Retail.Customer* data set. The data set *Retail.NewCustomer* contains 89954 observations and 12 variables.

The following table shows the variable names and formats in each SAS data set.

SAS Data Set	Variable Name	Variable Format
Retail.Customer	CountryBirth_Date	$COUNTRY.DATE9.
Retail.NewCustomer	Country_IDBirth_Date	$COUNTRY.MMDDYYP10

Techniques for changing the variable attributes of a SAS data set include

1. A DATA Step
2. PROCDATASETS.

The following sample programs show each of these techniques. You can use these samples as models for creating benchmark programs in your own environment. Your results might vary depending on the structure of your data, your operating environment, and the resources that are available at your site. You can also view general recommendations for eliminating unnecessary passes through the data.

Programming Techniques

---

❶ A DATA Step

This program uses a DATA step with a RENAME statement and a FORMAT statement to modify attributes for the variables **`Country_ID`** and **`Birth_Date`**.

```
data retail.newcustomer;
```

```
   set retail.newcustomer;
   rename Country_ID=country;
   format birth_date date9.;
run;
```

❷ PROC DATASETS

This program uses PROC DATASETS to modify the names and formats of the variables **Country_ID** and **Birth_Date.**

```
proc datasets lib=retail nolist;
   modify newcustomer;
   rename Country_ID=country;
   format birth_date date9.;
quit;
```

General Recommendations

- To save significant resources, use the DATASETS procedure with the NOLIST option instead of a DATA step to change the attributes of a SAS data set.

Reading and Writing Only Essential Data

Overview

Best practices specify that you should write programs that read and write only essential data. If you process fewer observations and variables, you conserve resources. This topic covers many different techniques that can improve performance when you

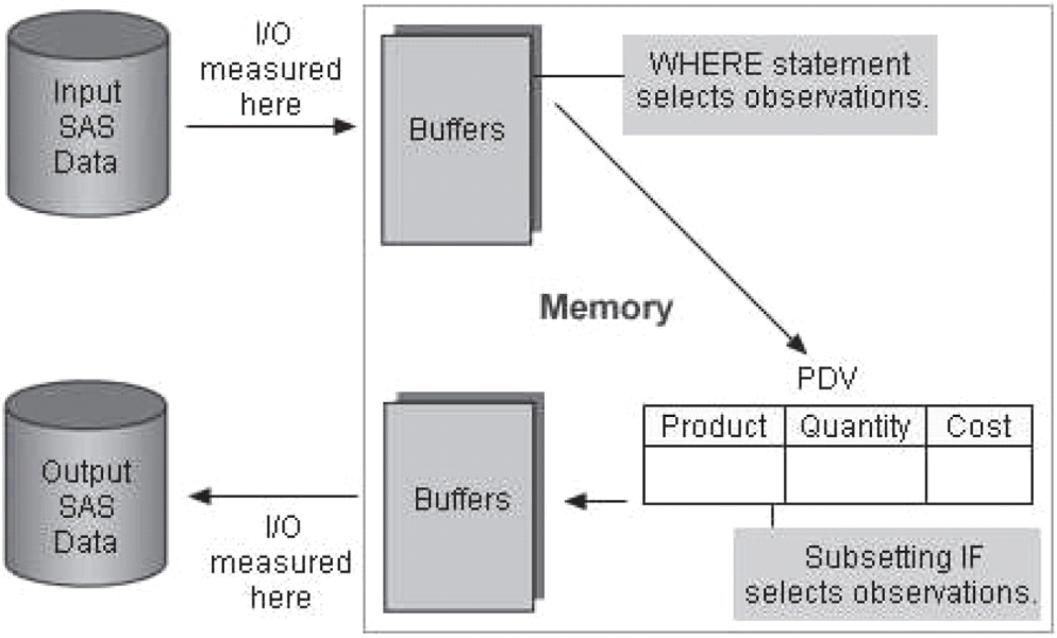
- select observations from SAS data sets
- select observations from external files
- keep or drop variables when creating or reporting on SAS data sets.

Look at how two useful statements differ in how they select observations to subset.

Selecting Observations Using Subsetting IF versus WHERE Statement

You can use WHERE statements or subsetting IF statements to subset data. Although both statements test a condition to determine whether SAS should process an observation, the WHERE statement is more efficient.

The following graphic illustrates differences in how these statements process data.



I/O operations are measured as data moves between the disk that contains input SAS data and the buffer in memory, and when data moves from the output buffer to the disk that contains output data sets. Input data is not affected by the WHERE statement or subsetting IF statement. However, output data is affected by both.

CPU time is measured when data must be processed in the program data vector. CPU time can be saved if fewer observations are processed.

A WHERE statement and an IF statement make different use of the program data vector. The WHERE statement examines what is in the input page buffer and selects observations before they are loaded in the program data vector, which results in a savings in CPU operations. The subsetting IF statement loads all observations sequentially into the program data vector. If the statement finds a match and the statement is true, then the data is processed and is written to the output page buffer.

WHERE statements work on variables that are already in SAS data sets. IF statements can work on any variable in the program data vector, including new or old variables.

Comparative Example: Creating a Subset of a SAS Data Set

Overview

Suppose you want to create a subset of the data set *Retail.Customer*. You want to include data for only the United Kingdom. The subset contains approximately 5.56% of the *Retail.Customer* data.

Techniques for subsetting observations include

1. Subsetting IF Statement
2. WHERE Statement.

The following sample programs show each of these techniques. You can use these samples as models for creating benchmark programs in your own environment. Your results might vary depending on the structure of your data, your operating environment, and the resources that are available at your site. You can also view general recommendations for reading and writing only essential data.

Programming Techniques

---

❶ Subsetting IF Statement

This program uses the IF statement to select observations if the value for **`Country`** is *GB*.

```
data retail.UnitedKingdom;
   set retail.customer;
   if country='GB';
run;
```

❷ WHERE Statement

This program uses the WHERE statement to select observations when the value for **Country** is *GB*. This can be more efficient than using a subsetting IF statement.

```
data retail.UnitedKingdom;
   set retail.customer;
   where country='GB';
run;
```

General Recommendations

- To save CPU resources when the subset is small, use a WHERE statement instead of a subsetting IF statement to subset a SAS data set.

Other Differences between the IF and WHERE Statements

Review the following table to note other differences between the IF and WHERE statements.

	The IF Statement	The WHERE Statement
Selecting Data	can select records from external files, observations from SAS data sets, observations created with an INPUT statement, or observations based on the value of a computed or derived variable.	can select only observations from SAS data sets.
Conditional Execution	is an executable statement	is not an executable statement
Grouping Data Using a BY Statement	produces the same data set when a BY statement accompanies a SET, MERGE, or UPDATE statement.	produces a different data set when a BY statement accompanies a SET, MERGE, or UPDATE statement.
Merging Data	selects observations after combining current observations.	applies the selection criteria to each input data set before combining observations.

Note If you use the WHERE= data set option and the WHERE statement in the same DATA step, SAS ignores the WHERE statement for input data sets. The WHERE= data set option and the WHERE statement call the same SAS routine.

Using the WHERE Statement with the OBS= and FIRSTOBS= Options

Another way to read and write only essential data is to process a segment of subsetted data. You accomplish this specialized task by using a WHERE expression in conjunction with the OBS= and FIRSTOBS= data set options.

In the following example, the WHERE expression selects observations before the OBS= and FIRSTOBS= options are applied. The values specified for OBS= and FIRSTOBS= are the logical observation numbers in the subset, not the physical observation numbers in the data set.

```
options fmtsearch=(formats);

proc print
     data=company.organization_dim(firstobs=5 obs=8);
     var employee_id employee_gender salary;
     where salary>40000;
run;
```

Obs	Employee_ID	Employee_Gender	Salary

101	120201	Male	$43.280
157	120257	Female	$156.245
158	120258	Male	$83.305
159	120259	Male	$433.800

FIRSTOBS = 5 is the fifth observation in the subset, whereas it was observation 101 in the data set *Company.Organization*.

OBS = 8 is the eighth observation in the subset, whereas it was observation 159 in the data set *Company.Company.Organization*.

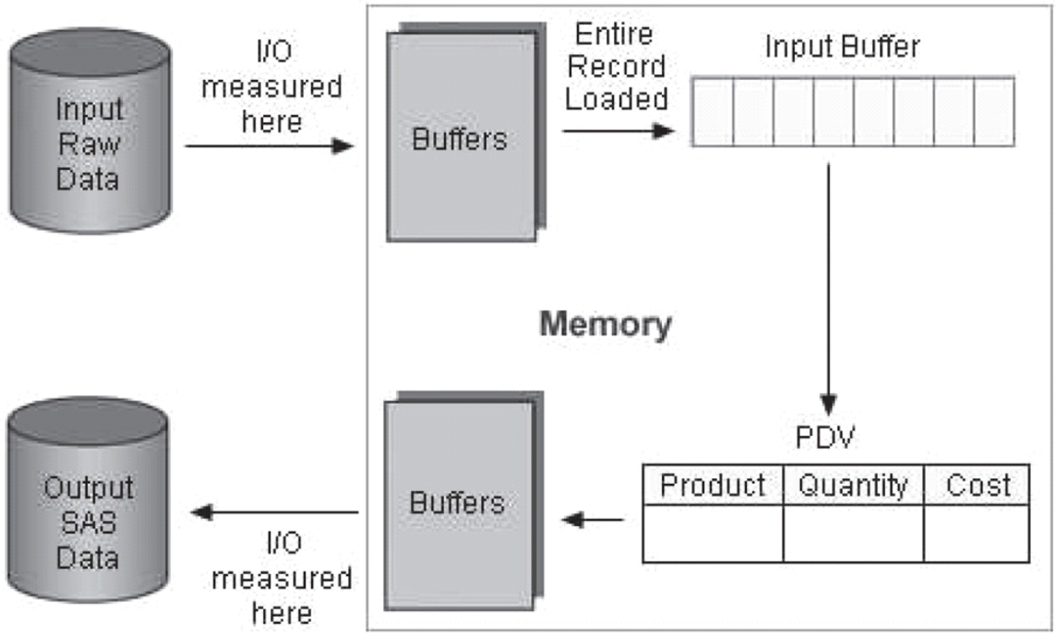
Now that you have seen techniques for efficiently subsetting observations that are read from SAS data sets, look at techniques for subsetting records that are read from external files.

Before viewing a comparative example that illustrates these techniques, it might be useful to review which resources are affected as SAS reads and processes data that is read from external files.

Selecting Observations When Reading Data from External Files

Positioning a subsetting IF statement in a DATA step so that it reads only the variables that are needed to select the subset—before reading all the data—can reduce the overhead required for processing data.

The following graphic illustrates how data is read from an external file, loaded into the input buffer, and read into the program data vector.



Remember that I/O operations are measured as data moves between disks and buffers— for both input and output data. All records are loaded into the input buffer before moving to the program data vector for processing, so I/O is not affected by the placement of a subsetting IF statement in the DATA step.

You can reduce the CPU resources that are required for processing data by limiting what is read into the program data vector. Position a subsetting IF statement after an INPUT statement that reads only the data that is required in order to check for specific conditions. Subsequent statements do not execute and do not process variable values for unwanted observations.

Note Converting raw character fields to SAS character variables requires less CPU time than converting raw numeric fields to the real binary encoding of SAS numeric variables.

Comparative Example: Creating a Subset of Data by Reading Data from an External File

Overview

Suppose you want to create a SAS data set by reading a subset of data from an external file that is referenced by the fileref *Customerdata*. You want the subset to contain only customers in the United Kingdom.

The subset is approximately 5.56% of the countries in the external file, which contains 89,954 records and 12 fields.

Techniques for doing this include

1. Reading All Variables and Subsetting
2. Reading Selected Variables and Subsetting.

The following sample programs show each of these techniques. You can use these samples as models for creating benchmark programs in your own environment. Your results might vary depending on the structure of your data, your operating environment, and the resources that are available at your site. You can also view general recommendations for reading and writing only essential data.

Programming Techniques

---

❶ Reading All Variables and Subsetting

In this program, the INPUT statement reads the values for all variables before the subsetting IF statement checks for the value of **`Country`.** Then, if the value for **`Country`** is ***`GB`***, the observation is written to the output data set *Retail.UnitedKingdom*.

```
data retail.UnitedKingdom;
   infile customerdata;
   input @1 Customer_ID         12.
         @13 Country            $2.
         @15 Gender             $1.
         @16 Personal_ID        $15.
         @31 Customer_Name      $40.
         @71 Customer_FirstName $20.
         @91 Customer_LastName  $30.
         @121 Birth_Date        date9.
         @130 Customer_Address  $45.
         @175 Street_ID         12.
         @199 Street_Number     $8.
         @207 Customer_Type_ID  8.;
if country='GB';
run;
```

❷ Reading Selected Variables and Subsetting

In this program, the first INPUT statement reads only the value for **`Country`** and holds the record in the input buffer using the single trailing @ sign. Then the program uses a subsetting IF statement to check for the value of **`Country`.** If the value for **`Country`** *is not GB*, values for other variables are not read in or written to the output data set *Retail.UnitedKingdom*. If the value for **`Country`** *is GB*, values for other variables are input and written to the output data set *Retail.UnitedKingdom*.

```
data retail.UnitedKingdom;
   infile customerdata;
   input @13 Country $2. @;
   if country='GB';
   input @1 Customer_ID              12.
         @15 Gender                  $1.
         @16 Personal_ID             $15.
         @31 Customer_Name           $40.
         @71 Customer_FirstName      $20.
         @91 Customer_LastName       $30.
         @121 Birth_Date            date9.
         @130 Customer_Address       $45.
         @175 Street_ID              12.
```

```
         @199 Street_Number          $8.
         @207 Customer_Type_ID        8.;
run;
```

General Recommendations

- Position a subsetting IF statement in a DATA step so that only variables that are necessary to select the record are read before subsetting. This can result in significant savings in CPU time. There is no difference in I/O or memory usage between the two techniques.
- When selecting rows of data from an external file, read the field or fields on which the selection is being made before reading all the fields into the program data vector.
- Use the single trailing @ sign to hold the input buffer so that you can continue to read the record when the variable or variables satisfy the IF condition.
- Reset the pointer so that you can begin reading the record in the first position but using **`@1 Customer_ID.`**

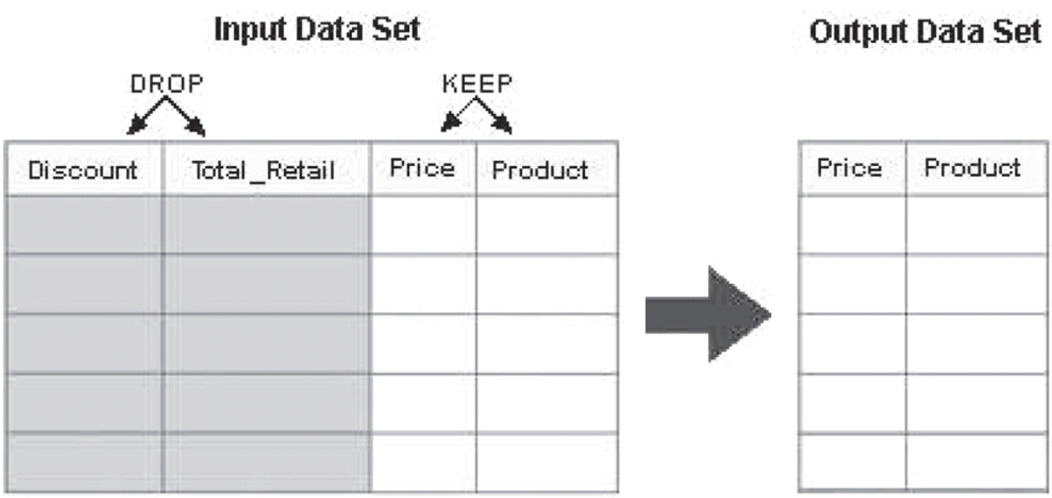
In addition to subsetting observations, you can subset variables by using statements or options that efficiently read and write only essential data.

Before viewing two comparative examples that illustrate how to best limit which variables are read and processed, review how these useful statements and options work.

Subsetting Variables with the KEEP= and DROP= Statements and Options

To subset variables, you can use

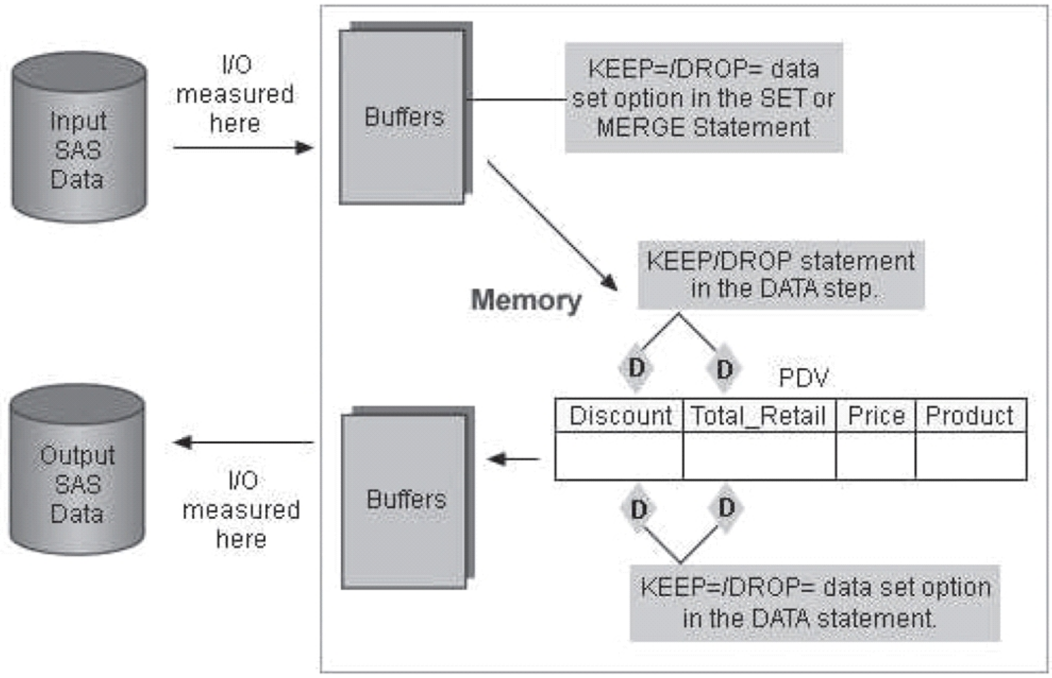
- the DROP and KEEP statements
- the DROP= and KEEP= data set options.



Note You cannot use the DROP and KEEP statements and the DROP= and KEEP= output data set options in the same step.

Use of the KEEP= data set option and the DROP= data set option can affect resource usage, depending on whether they are used in a SET or MERGE statement or in a DATA statement.

The following figure shows how options in these statements process data.



When used in the SET or MERGE statement, the KEEP= and DROP= data set options affect which variables are read into the program data vector. Reading only the variables that need to be processed in the DATA step can sometimes improve efficiency.

When used in the DATA statement, these same options put drop flags on variables to be excluded and affect which variables are written to the output data set.

The DROP and KEEP statements work just like the KEEP= or DROP= options in the DATA statement.

The following table describes differences in how the KEEP statement and the KEEP= data set option write variables to SAS data sets.

KEEP Statement	KEEP= Output Data Set Option	KEEP= Input Data Set Option
Causes a DATA step to write only the variables listed in the KEEP statement to one or more data sets.	Causes a DATA step to write only the variables listed in the KEEP= variable list to the output data set.	If there is no KEEP=/DROP= output data set option, causes the DATA step to write only the variables listed in the KEEP= variable list.
If there is no KEEP=/DROP= input data set option, enables the DATA step to process all of the variables.	If there is no KEEP=/DROP= input data set option, enables the DATA step to process all of the variables.	Enables processing of only the variables listed in the KEEP= variable list.
Applies to all data sets that are created within the same DATA step.	Can write different variables to different data sets.	If there is no KEEP=/DROP= output data set option or KEEP/DROP statement, enables processing of only the variables listed in the KEEP= variable list.
Available only in the DATA step	Available in the DATA step or in a PROC step.	Available in the DATA step or in a PROC step.

The following table describes differences in how the DROP statement and the DROP= data set option write variables to SAS data sets.

DROP Statement	DROP= Output Data Set Option	DROP= Input Data Set Option
Causes a DATA step to write only the variables not Usted in the DROP statement to one or more data sets.	Causes a DATA step to write only the variables not listed in the DROP= variable list to the output data set.	If there is no KEEP=/DROP= output data set option, enables processing of only the variables not listed in the DROP= variable list.
If there is no KEEP=/DROP= input data set option, enables the DATA step to process all of the variables.	If there is no KEEP=/DROP= input data set option, enables the DATA step to process all of the variables.	Enables processing of only the variables not listed in the DROP= variable list.

Applies to all data sets created within the same DATA step.	Can write different variables to different data sets.	If there is no KEEP=/DROP= output data set option or KEEP/DROP statement, enables processing of only the variables not listed in the DROP= variable list.
Available only in the DATA step.	Available in the DATA step or in a PROC step.	Available in the DATA step or in a PROC step.

Comparative Example: Creating a Report That Contains Average and Median Statistics

Overview

Suppose you want to create a report that contains the average and median values for the variable **`Profit`**, based on data that is read from the data set *Retail.Order_fact*. Depending on the number of variables eliminated, it might be more efficient to use the KEEP= option in a SET statement to limit which variables are read.

Techniques for reading and writing variables to a data set include

1. Without the KEEP= Statement
2. KEEP= in the DATA Statement
3. KEEP= in the DATA and SET Statements
4. KEEP= in the SET and MEANS Statements.

The following sample programs show each of these techniques. You can use these samples as models for creating benchmark programs in your own environment. Your results might vary depending on the structure of your data, your operating environment, and the resources that are available at your site. You can also view general recommendations for reading and writing only essential data.

Programming Techniques

---

❶ Without the KEEP= Statement

This program reads all variables from the data set *Retail.Order_fact* and does not restrict which variables are written to the output data set *Retail.Profit*. PROC MEANS reads all the variables from the data set.

```
data retail.profit;
   set retail.order_fact;
   if discount=. then
      Profit=(total_retail_price-costPrice_Per_Unit)*quantity;
   else Profit=((total_retail_price*discount)-costprice_per_unit)*quantity;
run;
proc means data=retail.profit mean median maxdec=2;
   title 'Order Information';
   class employee_id;
   var profit;
run;
```

❷ KEEP= in the DATA Statement

This program uses the KEEP= data set option in the DATA statement to write two variables to the output data set *Retail.Profit*. PROC MEANS reads only two variables from the data set.

```
data retail.profit (keep=employee_id profit);
   set retail.order_fact;
   if discount=. then
      Profit=(total_retail_price-costprice_per_unit)*quantity;
   else Profit=((total_retail_price*discount)-costprice_per_unit)*quantity;
run;
proc means data=retail.profit mean median maxdec=2;
   title 'Order Information';
   class employee_id;
   var profit;
run;
```

❸ KEEP= in the DATA and SET Statements

This program uses the KEEP= option in the SET statement to read six variables from *Retail.Order_fact*, and it uses the KEEP= data set option in the DATA statement to write two variables to the output data set *Retail.Profits*. PROC MEANS reads only two variables from the data set.

```
data retail.profiti (keep=employee_id profit);
   set retail.order_fact (keep=employee_idtotal_retail_price discount
                               costprice_per_unit quantity);
      if discount=. then
         Profit=(total_retail_price-costprice_per_unit)*quantity;
      else Profit=((total_retail_price*discount)-costprice_per_unit)*quantity;
run;
proc means data=retail.profit mean median maxdec=2;
   title 'Order Information';
   class employee_id;
   var profit; run;
```

❹ KEEP= in the SET and MEANS Statements

This program uses the KEEP= option in the SET statement to read selected variables from *Retail.Order_fact*, and it uses the KEEP= data set option in the MEANS statement to process only the variables that are needed for the statistical report. You might do this if you need additional variables in *Retail.Profits* for further processing, but only two variables for processing by PROC MEANS.

```
data retail.profit;
   set retail.order_fact (keep=employee_idtotal_retail_price discount
                               costprice_per_unit quantity);
   if discount=. then
      Profit=(total_retail_price-costprice_per_unit)*quantity;
   else Profit=((total_retail_price*discount)-costprice_per_unit)*quantity;
run;
proc means data=retail.profit (keep=employee_id profit) mean median maxdec=2;
   title 'Order Information';
   class employee_id;
   var profit;
run;
```

General Recommendations

- To reduce both CPU time and I/O operations, avoid reading and writing variables that are not needed.

Comparative Example: Creating a SAS Data Set That Contains Only Certain Variables

Overview

Suppose you want to read data from an external file that is referenced by the fileref *Rawdata* and to create a SAS data set that contains only the variables **`Customer_ID, Country, Gender`**, and **`Customer_Name.`**

Techniques for accomplishing this task include

1. Reading All Fields
2. Reading Selected Fields.

The following sample programs show each of these techniques. You can use these samples as models for creating benchmark programs in your own environment. Your results might vary depending on the structure of your data, your operating environment, and the resources that are available at your site. You can also view general recommendations for reading and writing only essential data.

Programming Techniques

---

❶ Reading All Fields

In this program, the KEEP= data set option writes only the variables that are needed to the output data set, whereas the INPUT statement reads all fields from the external file.

```
input @1   Customer_ID        12.
      @13  Country            $2.
      @15  Gender             $1.
      @16  Personal_ID        $15
      @31  Customer_Name      $40.
      @71  Customer_FirstName $20.
      @91  Customer LastName  $30.
      @121 Birth_Date        date9.
      @130 Customer Address  $45.
      @175 Street_ID         12.
      @199 Street Number     $8.
      @207 Customer _Type_ID 8.;
run;
```

❷ Reading Selected Fields

In this program, the INPUT statement reads selected fields from the external file, and by default, these are written to the output data set.

```
data retail.customers;
   infile rawdata;
   input @1  Customer_ID    12.
         @13 Country       $2.
         @15 Gender        $1.
         @31 Customer Name $40.;
run;
```

General Recommendations

- When possible, read only the fields you need from an external data file to save CPU and real-time resources.
- To save CPU resources, avoid converting numerics that you do not need in further processing.

Note Remember that numeric data is moved into the program data vector after being converted to real binary, floating point numbers; multiple digits are stored in one byte. Character data is moved into the program data vector with no conversion; one character is stored in one byte.

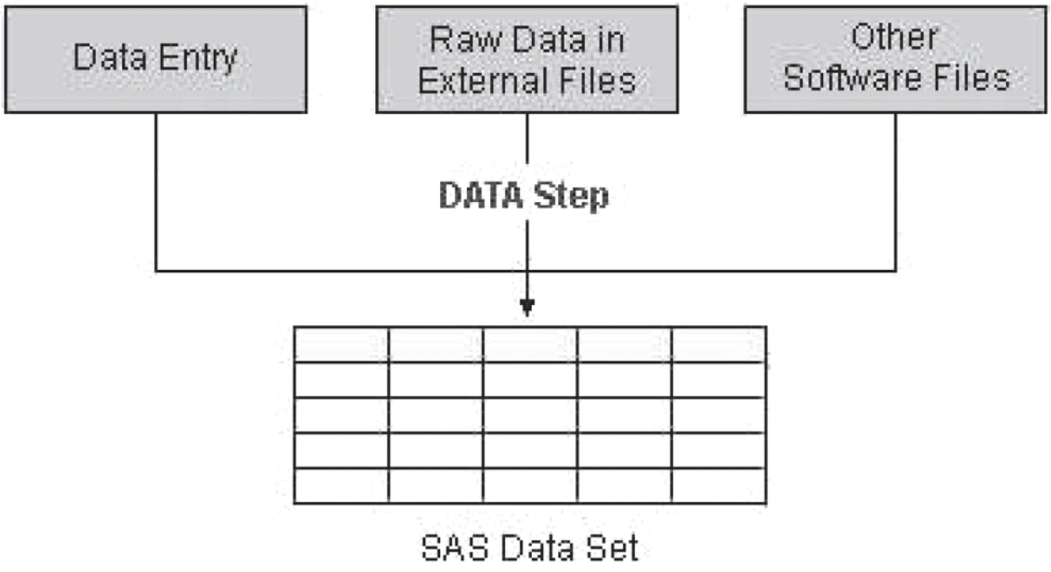
Storing Data in SAS Data Sets

Overview

In many cases, it is best practice for you to store data in SAS data sets. You can optimize performance if you know when you should create a SAS data set and when you should read data directly from an external file.

Before viewing the comparative example that illustrates different techniques for reading from a SAS data set versus from an external file, consider the following advantages of storing data in SAS data sets.

When you use SAS to repeatedly analyze or manipulate any particular group of data, it is more efficient to create a SAS data set than to read the raw data each time. Although SAS data sets can be larger than external files and can require more disk space, reading from SAS data sets saves CPU time that is associated with reading a raw data file.



Other reasons for storing data in SAS data sets, rather than external files, include:

- When the data is already in a SAS data set, you can use a SAS procedure, function, or routine on the data without further conversion.
- SAS data sets are self-documenting.

The descriptor portion of a SAS data set documents information about the data set such as

- data set labels
- variable labels
- variable formats
- informats
- descriptive variable names.

Note Create a temporary SAS data set if the data set is used for intermediate tasks such as merging and if it is needed in that SAS session only. Create a temporary SAS data set when the external file on which the data set is based might change between SAS sessions.

Comparative Example: Creating a SAS Data Set or Reading from an External File

Overview

Suppose you want to create a SAS data set that contains a large number of variables. One way to accomplish this task is to read from raw data fields in an external file that is referenced by the fileref *Rawdata*. Another way to accomplish this is to read the same data values from an existing SAS data set named *Retail.Customer*.

Techniques for accomplishing this task include

1. Reading from an External File
2. Reading from a SAS Data Set.

The following sample programs show each of these techniques. You can use these samples as models for creating benchmark programs in your own environment. Your results might vary depending on the structure of your data, your operating environment, and the resources that are available at your site. You can also view general recommendations for storing data in SAS data sets.

Programming Techniques

❶ Reading from an External File

In this program, the INPUT statement reads fields of data from an external file that is referenced by the fileref *Rawdata* and creates 12 variables. For benchmarking purposes, the DATA statement creates a _NULL_ data set, testing for the effects of the reading operation rather than the output processing.

```
data _null_;
   infile rawdata;
   input @1   Customer_ID        12.
         @13  Country            $2.
         @15  Gender             $1.
         @16  Personal_ID        $15.
         @31  Customer Name      $40.
         @71  Customer FirstName $20.
         @91  Customer LastName  $30.
         @121 Birth_Date         date9.
         @130 Customer Address   $45.
         @175 Street_ID          12.
         @199 Street Number      $8.
         @207 Customer _Type_ID  8.;
run;
```

❷ Reading from a SAS Data Set

In this program, the SET statement reads data directly from an existing SAS data set. As in the previous program, the DATA statement uses _NULL_ instead of naming a data set.

```
data _null_;
   set retail.customer;
run;
```

General Recommendations

- To save CPU resources, you can read variables from a SAS data set instead of reading data from an external file.
- To reduce I/O operations, you can read variables from a SAS data set instead of reading data from an external file. However, savings in I/O operations are largely dependent on the block size of the external data file and on the page size of the SAS data set.

Avoiding Unnecessary Procedure Invocation

Overview

Best practices specify that you avoid unnecessary procedure invocation. One way to do this is to take advantage of procedures that accomplish multiple tasks with one invocation.

Several procedures enable you to create multiple reports by invoking the procedure only once. These include

- the SQL procedure
- the DATASETS procedure
- the FREQ procedure
- the TABULATE procedure.

Note BY-group processing can also minimize unnecessary invocations of procedures. To illustrate this principle, examine features of the DATASETS procedure.

Executing the DATASETS Procedure

The DATASETS procedure can use RUN-group processing to process multiple sets of statements. RUN-group processing enables you to submit groups of statements without ending the procedure.

When the DATASETS procedure executes,

- SAS reads the program statements that are associated with one task until it reaches a RUN statement or an implied RUN statement.
- SAS executes all of the preceding statements immediately, and then continues reading until it reaches another RUN statement or an implied RUN statement.

To execute the last task, you must use a RUN statement or a QUIT statement.

```
proc datasets lib=company;
   modify orders;
      rename quantity=Units_Ordered;
      format costprice_per_unit dollar13.2;
      label delivery_date='Date of Delivery';
 run;
 modify customers;
   format customer_birthdate mmddyy10.
 run;
quit;
```

You can terminate the PROC DATASETS execution by submitting

- a DATA statement
- a PROC statement
- a QUIT statement.

RUN-Group Processing

If you can take advantage of RUN-group processing, you can avoid unnecessary procedure invocation. For best programming practices, you need to understand how RUN-group processing affects the execution of SAS statements. The procedures that support RUN-group processing include

- CHART, GCHART
- PLOT, GPLOT
- GIS, GMAP
- GLM
- REG
- DATASETS.

Using Different Types of RUN Groups with PROC DATASETS

To illustrate how RUN-group processing works, this discussion focuses on the DATASETS procedure. The comparative example that follows includes programs that use PROC DATASETS to modify the descriptor portion of data sets. Before you examine the code to consider efficient programming techniques, review how the principles associated with RUN-group processing apply to PROC DATASETS.

The DATASETS procedure supports four types of RUN groups. Each RUN group is defined by the statements that compose it and by what causes it to execute.

Some statements in PROC DATASETS act as implied RUN statements because they cause the RUN group that precedes them to execute.

The following list identifies which statements compose a RUN group and what causes each RUN group to execute:

- The PROC DATASETS statement always executes immediately. No other statement is necessary to cause the PROC DATASETS statement to execute. Therefore, the PROC DATASETS statement alone is a RUN group.
- The MODIFY statement and any of its subordinate statements form a RUN group. These RUN groups always execute immediately. No other statement is necessary to cause a MODIFY RUN group to execute.

- The APPEND, CONTENTS, and COPY statements (including EXCLUDE and SELECT, if present) form their own separate RUN groups. Every APPEND statement forms a single-statement RUN group; every CONTENTS statement forms a single-statement RUN group; and every COPY step forms a RUN group. Any other statement in the procedure, except those that are subordinate to either the COPY or MODIFY statement, causes the RUN group to execute.

Additionally, one or more of the following statements form a RUN group:

- AGE
- EXCHANGE
- CHANGE
- REPAIR.

If any of these statements appear in sequence in the PROC step, the sequence forms a RUN group. For example, if a REPAIR statement appears immediately after a SAVE statement, the REPAIR statement does not force the SAVE statement to execute; it becomes part of the same RUN group. To execute the RUN group, submit one of the following statements:

- PROC DATASETS
- MODIFY
- APPEND
- QUIT
- CONTENTS
- RUN
- COPY
- another DATA or PROC step.

Comparative Example: Modifying the Descriptor Portion of SAS Data Sets

Overview

Suppose you want to use the DATASETS procedure to modify the data sets *NewCustomer, NewOrders*, and *NewItems*.

Techniques for accomplishing this task include using

1. Multiple DATASETS Procedures
2. A Single DATASETS Procedure.

The following sample programs show each of these techniques. You can use these samples as models for creating benchmark programs in your own environment. Your results might vary depending on the structure of your data, your operating environment, and the resources that are available at your site. You can also view general recommendations for avoiding unnecessary procedure invocation.

Programming Techniques

---

❶ Multiple DATASETS Procedures

This program invokes PROC DATASETS three times to modify the descriptor portion of the data set *NewCustomer*, two times to modify the descriptor portion of the data set *NewOrders*, and once to change the name of the data set *Newitems*.

```
proc datasets lib=company nolist;
   modify newcustomer;
   rename Country_ID=Country
```

```
          Name=Customer_Name;
quit;

proc datasets lib=company nolist;
   modify newcustomer;
   format birth_date date9.;
quit;

proc datasets lib=company nolist;
   modify newcustomer;
   label birth_date='Date of Birth';
quit;

proc datasets lib=company nolist;
   modify neworders;
   rename order=Order_ID
   employee=Employee_ID
   customer=Customer_ID;
quit;

proc datasets lib=company nolist;
   modify neworders;
   format order_date date9.;
quit;

proc datasets lib=company nolist;
   change newitems=NewOrder_Items;
quit
```

❷ Single DATASETS Procedure

This program invokes PROC DATASETS once to modify the descriptor portion of the data sets *NewCustomer* and *NewOrders*, and to change the name of the data set *Newitems*. This technique is more efficient.

```
proc datasets lib=company nolist;
   modify newcustomer;
   rename country_ID=Country
          name=Customer_Name;
   format birth_date date9.;
   label birth_date='Date of Birth';
   modify neworders;
   rename order=Order_ID
          employee=Employee_ID
          customer=Customer_ID;
   format order_date date9.;
   change newitems=NewOrder_Items;
quit;
```

General Recommendations

- Invoke the DATASETS procedure once and process all the changes for a library in one step to save CPU and I/O resources—at the cost of memory resources.
- Use the NOLIST option on the PROC DATASETS statement. The NOLIST option suppresses printing of the library members in the log. Using NOLIST can save I/O.

Note Because the specified library could change between invocations of the DATASETS procedure, the procedure is reloaded into memory for each invocation.

Summary

Executing Only Necessary Statements

You minimize the CPU time that SAS uses when you execute the minimum number of statements in the most efficient order.

For a more efficient program, place the subsetting IF statement as soon as logically possible in a DATA step when creating

a subset of data.

Review guidelines for using conditional logic efficiently with IF-THEN/ELSE statements or SELECT statements. Remember to minimize the number of statements that use SAS functions or arithmetic operators.

Conditionally execute only necessary statements by placing statements in groups that are associated with IF-THEN/ELSE statements or SELECT/WHEN statements. Groups of statements execute only when a particular condition is true. Review the criteria for using DO groups efficiently.

Review the related comparative examples:

- "Comparative Example: Creating a Subset of Data" on page 768
- "Comparative Example: Creating Variables Conditionally Using DO Groups" on page 771
- "Comparative Example: Creating Variables Conditionally When Calling Functions" on page 773
- "Comparative Example: Creating Data in DO Groups" on page 778

Eliminating Unnecessary Passes through the Data

You should avoid reading or writing data more than necessary in order to minimize I/O operations.

There are a variety of techniques that you can use. For example, use a single DATA step to create multiple output data sets from one pass of the input data, rather than using multiple DATA steps to process the input data each time that you create an output data set. Create sorted subsets by subsetting data with the SORT procedure rather than subsetting data in a DATA step and then sorting. Change variable attributes by using PROC DATASETS rather than a DATA step.

Review the related comparative examples:

- "Comparative Example: Creating Multiple Subsets of a SAS Data Set" on page 782
- "Comparative Example: Creating a Sorted Subset of a SAS Data Set" on page 784
- "Comparative Example: Changing the Variable Attributes of a SAS Data Set" on page 785

Reading and Writing Only Essential Data

If you process fewer observations and variables, SAS performs fewer I/O operations. To limit the number of observations that are processed, you can use the subsetting IF statement and the WHERE statement. Best programming practices can be applied if you understand other differences between subsetting IF and WHERE statements. You can also improve performance by applying OBS= and FIRSTOBS= processing with a WHERE statement.

To select observations when reading data from external tiles, position a subsetting IF statement in a DATA step so that it reads only the variables that are needed to select the subset before reading all the data. This can reduce the overhead required to process data.

To limit the number of variables that are processed, you can use

- the DROP and KEEP statements
- the DROP= and KEEP= data set options.

In the SET statement, the DROP= or KEEP= data set option controls which variables are read and subsequently processed. In the DATA statement, the DROP= or KEEP= data set option controls which variables are written to a data set after processing. Using the SET statement with these options is the most efficient and best practice.

Review the related comparative examples:

- "Comparative Example: Creating a Subset of a SAS Data Set" on page 787
- "Comparative Example: Creating a Subset of Data by Reading Data from an External File" on page 790
- "Comparative Example: Creating a Report That Contains Average and Median Statistics" on page 794

- "Comparative Example: Creating a SAS Data Set That Contains Only Certain Variables" on page 796

Storing SAS Data in SAS Data Sets

When you use SAS to repeatedly analyze or manipulate any particular group of data, create a SAS data set instead of reading the raw data each time.

Reading data from an external file versus reading from a SAS data set greatly increases CPU usage.

Review the related comparative example:

- "Comparative Example: Creating a SAS Data Set or Reading from an External File" on page 798

Avoiding Unnecessary Procedure Invocation

Invoking procedures once rather than multiple times can be the most efficient way to process data. Several procedures enable you to create multiple reports by invoking the procedure only once.

Using a single DATASETS procedure instead of multiple DATASETS procedures to modify the descriptor portion of a data set results in a noticeable savings in both CPU and I/O operations. Also, you can take advantage of RUN-group processing to submit groups of statements without ending the procedure.

Review the related comparative example:

- "Comparative Example: Modifying the Descriptor Portion of SAS Data Sets" on page 802

Quiz

Select the best answer for each question. After completing the quiz, check your answers using the answer key in the appendix.

1. Placing the subsetting IF statement at the top rather than near the bottom of a DATA step results in a savings in CPU usage. What happens if the subset is large rather than small? ?
   a. The savings in CPU usage increases as the subset grows larger because the I/O increases.
   b. The savings in CPU usage decreases as the subset grows larger. However, placing the subsetting IF statement at the top of a DATA step always uses fewer resources than placing it at the bottom.
   c. The savings in CPU usage remains constant as the subset grows larger. However, placing the subsetting IF statement near the bottom of a data set is preferable.
   d. The savings in CPU usage decreases as the subset grows larger. However, placing the subsetting IF statement near the bottom of a data set increases the I/O.
2. Which of the following statements is true about techniques that are used for modifying data and attributes? ?
   a. You can use PROC DATASETS to modify both data values and variable attributes.
   b. You can use PROC DATASETS to modify only data values.
   c. You can use the DATA step to modify both data values and variable attributes.
   d. You can use the DATA step to modify only variable attributes.
3. For selecting observations, is a subsetting IF statement or a WHERE statement more efficient? Why? ?
   a. A subsetting IF statement is more efficient because it loads all observations sequentially into the program data vector.
   b. A subsetting IF statement is more efficient because it examines what is in the input buffer and selects observations before they are loaded into the program data vector, which results in a savings in CPU operations.
   c. A WHERE statement is more efficient because it loads all observations sequentially into the program data vector.

d. A WHERE statement is more efficient because it examines what is in the input page buffer and selects observations before they are loaded into the program data vector, which results in a savings in CPU operations.

4. When is it more advantageous to create a temporary SAS data set rather than a permanent SAS data set? ?

a. When the external file on which the data set is based might change between SAS sessions.

b. When the external file on which the data set is based does not change between SAS sessions.

c. When the data set is needed for more than one SAS session.

d. When you are converting raw numeric values to SAS data values.

5. When you compare the technique of using multiple DATASETS procedures to using a single DATASETS procedure to modify the descriptor portion of a data set, which is true? ?

a. A one-step DATASETS procedure results in an increase in I/O operations.

b. Multiple DATASETS procedures result in a decrease in I/O operations.

c. A one-step DATASETS procedure results in a decrease in CPU usage.

d. Multiple DATASETS procedures result in a decrease in CPU usage.

Answers

1. Correct answer: b

As SAS processes a larger subset of the data, more CPU resources are required. However, positioning of the subsetting IF statement in a DATA step can affect performance and efficiency.

2. Correct answer: a

The DATA step is the only technique that can be used to modify both data values and variable attributes. The DATASETS procedure enables you to modify only variable attributes.

3. Correct answer: d

For selecting observations, a WHERE statement is more efficient than a subsetting IF statement because it examines what is in the input buffer and selects observations before they are loaded into the program data vector, which results in a savings in CPU operations.

4. Correct answer: a

It is more advantageous to create a temporary SAS data set rather than a permanent SAS data set when the external file on which the data set is based is frequently updated between SAS sessions.

5. Correct answer: c

A one-step DATASETS procedure results in a savings of CPU usage and I/O operations. PROC DATASETS supports RUN-group processing, which enables to process multiple SAS data sets from the same library with one invocation of the procedure.